C++进阶

运算符重载

1. 运算符重载，就是对已有的运算符重新进行定义，赋予其另一种功能，简化操作 让已有的运算符适应不同的数据类型。
2. 运算符重载其实就是定义一个函数，在函数体内实现想要的功能，当用到该运算符时，编译器会自动调用这个函数。也就是说，运算符重载是通过函数实现的，它本质上是函数重载。
3. **语法：**函数的名字由关键字operator及其紧跟的运算符组成，比如：重载+运算符 ==>operator+ 重载=号运算 ==>operator =
4. **注意：**重载运算符 不要更改 运算符的本质操作（+是数据的相加 不要重载成相减）。

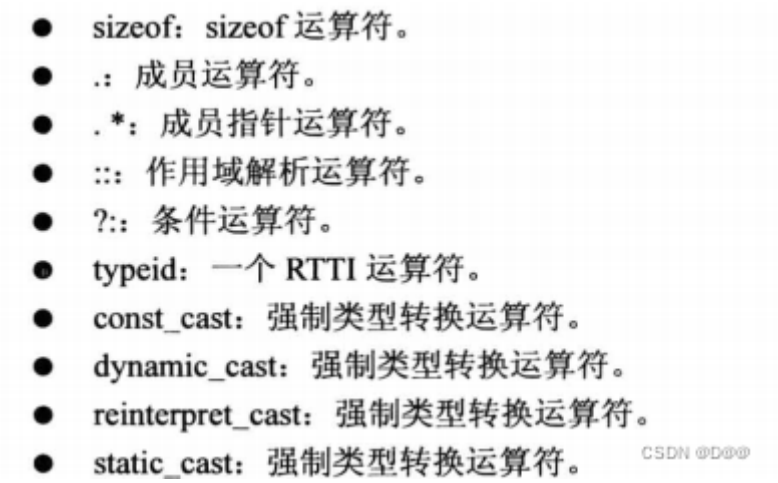
重载的限制

多数C++运算符都可以重载，重载的运算符不必是成员函数，但必须至少有一个操作数是用户定义的类型。

1. 重载后的运算符必须至少有 一 个操作数是用户定义的类型，防止用户为标准类型重载运算符。如：不能将减法运算符(-)重载为计算两个 double 值的和，而不是它们的差。虽然这种限制将对创造性有所影响，但可以确保程序正常运行。
2. 使用运算符时不能违反运算符原来的句法规则。例如，不能将二元操作数求模运算符(%)重载成使用一个操作数：

   ```
   int x;Time shiva;
   %x;
   %shiva;,
   ```

   且不能修改运算符的优先级。
3. 不能创建新运算符。例如，不能定义operator **()函数来表示求幂。
4. 不能重载以下运算符

5. 可以重载以下运算符

可重载的运算符

+	-	*	/	%	^
&	|	~=	!	=	<
>	+=	-=	*=	/=	%=

续表

^=	&=	|=	<<	>>	>>=
<<=	==	!=	<=	>=	&&
||	++	—	,	->*	->
()	[]	new	delete	new []	delete []

如何重载?

```
#include<iostream>

using namespace std;

struct node{
    int x,y;
    node operator + (node p){ //这是两个操作数的，自己本身加上另一个操作数
        node a;
        a.x = x + p.x;
        a.y = y + p.y;
        return a;
    }//相当于一个函数，只不过要写上operator 重载的符号，然后重载的规则

    node operator < (node p){
        if(x == p.x){
            if(y > p.y) return *this;
            else return p;
        }else if(x > p.x) return *this;
        else return p;
    }//注意：因为我们这个运算符是在类中写的，所以是通过对象调用的，那么this指针会占一个参数，而且是第一个参数，也就是说我们重载一个运算符，是在类中，而这个运算符是个单元运算符，那么参数列表就不用写东西了，是双元运算符，那么就需要传另一个参数进来
    friend node operator > (node a,node b){ //倘若是有前面加了friend的就写两个变量，因为使用了friend就不能使用this指针
        if(a.x > b.x) return a;
        else return b;
    }
};

int main(){

    node x,y,t;
    x.x = 1;
    x.y = 1;
    y.x = 2;
    y.y = 3;
    t.x = 5;
    t.y = 6;
    node s = x + y + t;
    printf("%d %d\n",s.x,s.y);

    s = x < y;
    printf("%d %d\n",s.x,s.y);
```

```
    s = x > y;
    printf("%d %d",s.x,s.y);
    return 0;
}
```

重载的妙用

在我们算法竞赛中，重载符一般用来排序使用的，当我们使用sort函数给结构体排序的时候，倘若我们没有写第三方排序函数，他就会遵循我们结构体的 < 来排序，倘若我们重载了，他就会遵循重载后的排序。
倘若我们使用重载来给sort排序的时候，我们重载符号一定要遵循：返回值一定要为 bool 值，因为是制定这样返回到底是真是假的规则

```
#include<iostream>
#include<algorithm>

using namespace std;

struct node{
    int x,y;
    bool operator < (node p){
        return x < p.x;
    }

}a[100];

int main(){

    a[1] = {4,2};
    a[2] = {2,3};
    sort(a + 1,a + 2 + 1); //x作为关键词，从小到大排序

    for(int i = 1; i <= 2; i++){
        printf("%d ",a[i].x);
    }

    return 0;
}
```

进阶排序

归并排序:

归并排序：基于分治的思想
算法步骤:对于排序 1 ~ n我们可以先排序 1 ~ n / 2,n / 2 + 1 ~ n以此类推再合并
稳定性: 稳定
复杂度：$O(nlogn)$
空间复杂度 : $O(n)$

```
void merge(int l, int r) {
  if (r - l <= 1) return;
  int mid = l + ((r - l) >> 1);
  merge(l, mid), merge(mid, r);
  for (int i = l, j = mid, k = l; k < r; ++k) {
    if (j == r || (i < mid && a[i] <= a[j]))
      tmp[k] = a[i++];
    else
      tmp[k] = a[j++];
  }
  for (int i = l; i < r; ++i) a[i] = tmp[i];
}
```

思考:能不能在交换途中求出逆序对个数?

```
//a数组是需要排序的数组，tmp数组是临时数组
void merge(int l, int r) {
  if (r - l <= 1) return;
  int mid = l + ((r - l) >> 1);
  merge(l, mid), merge(mid, r);
  for (int i = l, j = mid, k = l; k < r; ++k) {
    if (j == r || (i < mid && a[i] <= a[j]))
      tmp[k] = a[i++];
    else{
        cnt += mid - i;
        tmp[k] = a[j++];
    }
  }
  for (int i = l; i < r; ++i) a[i] = tmp[i];
}
```

快速排序

快速排序：一个基于分治思想的排序
算法步骤:在当前序列中确定一个基准数，使得当前序列相对有序。
稳定性:不稳定
时间复杂度:$O(nlogn)$ 最坏复杂度$O(n^2)$ 最坏复杂度举例: 1 2 3 4 5 6 7 8
空间复杂度:$O(n)$

```
void quick_sort(int q[], int l, int r) {
    if (l >= r) return;

    int x = q[l + r >> 1], i = l - 1, j = r + 1;
    while (i < j) {  //前面的 do防止死循环
        do i ++ ; while (q[i] < x); //找到第一个小于x的
        do j -- ; while (q[j] > x); //找到第一个大于x的
        if (i < j) swap(q[i], q[j]);//交换
    }

    quick_sort(q, l, j);
    quick_sort(q, j + 1, r);
}
```

快速排序优化:
1.通过 三数取中（即选取第一个、最后一个以及中间的元素中的中位数） 的方法来选择两个子序列的分界元素（即比较基准）。这样可以避免极端数据（如升序序列或降序序列）带来的退化；
2.当序列较短时，使用 插入排序 的效率更高；
3.每趟排序后，将与分界元素相等的元素聚集在分界元素周围，这样可以避免极端数据（如序列中大部分元素都相等）带来的退化。

```
template <typename T>
// arr 为需要被排序的数组，len 为数组长度
void quick_sort(T arr[], const int len) {
  if (len <= 1) return;
  // 随机选择基准（pivot）
  const T pivot = arr[rand() % len];
  // i：当前操作的元素
  // j：第一个等于 pivot 的元素
  // k：第一个大于 pivot 的元素
  int i = 0, j = 0, k = len;
  // 完成一趟三路快排，将序列分为：
  // 小于 pivot 的元素 | 等于 pivot 的元素 | 大于 pivot 的元素
  // 每次把大于pivot的数从最后边开始放，
  while (i < k) { //如果当前操作数 >= k 因为k是第一个大于pivot的数，所以没必要比较，所以j ~ i的数一定都是pivot
    if (arr[i] < pivot) //如果当前的数比pivot小的话，我们就把他与第一个等于pivot的交换。
      swap(arr[i++], arr[j++]); //交换后我们明确 1~j都是小于pivot的数，j + 1~i是等于pivot的数
    else if (pivot < arr[i]) //如果当前操作数大于pivot我们就与k位置交换
      swap(arr[i], arr[--k]); //交换后我们并不知道交换过来的数与pivot的关系
    else  //如果相等直接++
      i++;
  }
  // 递归完成对于两个子序列的快速排序
  quick_sort(arr, j);
  quick_sort(arr + k, len - k);
}
```

思考:能不能在交换途中求出逆序对个数?

求第k大值

基于快速排序的思想
算法步骤：每次确定一个基准点，小于基准点的放在一个数组里，元素个数为x，等于基准点的放在一个数组里，元素个数为y，大于基准点的放在一个数组里，元素个数为z，若k大于 x + y那么 一定在x + y + 1 ~ z里面，否则一定在1 ~ x + y里面
算法复杂度: 最优复杂度$O(n)$，最坏复杂度$O(n^2)$
空间复杂度: $O(nlogn)$

```
template <typename T>
T find_kth_element(T arr[], int rk, const int len) {
  if (len <= 1) return arr[0];
  // 随机选择基准（pivot）
  const T pivot = arr[rand() % len];
  // i：当前操作的元素
  // j：第一个等于 pivot 的元素
  // k：第一个大于 pivot 的元素
```

```
  int i = 0, j = 0, k = len;
  // 完成一趟三路快排，将序列分为：
  // 小于 pivot 的元素 | 等于 pivot 的元素 | 大于 pivot 的元素
  // 每次把大于pivot的数从最后边开始放，
  while (i < k) { //如果当前操作数 >= k 因为k是第一个大于pivot的数，所以没必要比较，所以j ~ i的数一定都是pivot
    if (arr[i] < pivot) //如果当前的数比pivot小的话，我们就把他与第一个等于pivot的交换。
      swap(arr[i++], arr[j++]); //交换后我们明确 1~j都是小于pivot的数，j + 1~i是等于pivot的数
    else if (pivot < arr[i]) //如果当前操作数大于pivot我们就与k位置交换
      swap(arr[i], arr[--k]); //交换后我们并不知道交换过来的数与pivot的关系
    else  //如果相等直接++
      i++;
  }
  if (rk < j) return find_kth_element(arr, rk, j); //如果排名 < j，那就往左边找
  else if (rk >= k)
    return find_kth_element(arr + k, rk - k, len - k); //如果排名 >= k那么左半边都不看了，所以总排名 - k
  return pivot;
}
```

计数排序:

计数排序：一种线性的排序算法
算法步骤:对原数组a来说建立一个新的数组b，$b[i]$表示的是 a数组中元素大小为i的个数，随后再创建一个数组c，$c[i]$表示的是$b[1]$ $b[i]$的和。那么$c[i]-c[i-1]$就是 a数组中元素大小为i的个数，并且我们知道 这个数排序后的下标在$c[i-1]+1-c[i]$
稳定性：稳定
时间复杂度: $O(n+w)$ //w是值域
空间复杂度 $O(n+w)$

```
void counting_sort() {
  memset(cnt, 0, sizeof(cnt));
  for (int i = 1; i <= n; ++i) ++cnt[a[i]];
  for (int i = 1; i <= w; ++i) cnt[i] += cnt[i - 1];
  for (int i = n; i >= 1; --i) b[cnt[a[i]]--] = a[i];
}
```

思考:能不能在交换途中求出逆序对个数?

基数排序

定义：基数排序是基于位来选择其他排序算法进行排序的
算法步骤:分别将位数设置第一个关键字，第二关键字..第k关键词，然后先排序第一关键字再排序第二关键字....
空间复杂度 $O(n)$
时间复杂度 $O(k/x*(n+pow(10,x))$ k是位数，x是几位为一组
稳定性:稳定

```
#include <bits/stdc++.h>
```

```
using namespace std;
#define int long long
const int maxn = 5e6 + 5;
int n, a[maxn], maxx, k;
vector<int> v[maxn], v2[maxn];
signed main(){
    cin >> n;
    for (int i = 1; i <= n; i++){
        cin >> a[i];
        maxx = max(maxx, a[i]);
    }
    while (maxx){
        k++;
        maxx /= 10;
    }
    for (int i = 1; i <= n; i++){
        for (int j = k; j >= 1; j--){
            v[i].push_back((a[i] / (int)pow(10, j - 1)) % 10);
        }
    }
    for (int j = k - 1; j >= 0; j--){
        int cnt[10] = {};
        for (int i = 1; i <= n; i++){
            cnt[v[i][j]]++;
        }
        for (int i = 1; i <= 10; i++){
            cnt[i] += cnt[i - 1];
        }
        for (int i = n; i >= 1; i--){
            v2[cnt[v[i][j]]--] = v[i];
        }
        for (int i = 1; i <= n; i++){
            v[i] = v2[i];
        }
    }
    for (int i = 1; i <= n; i++){
        for (int j = 0; j < v[i].size(); j++){
            cout << v[i][j];
        }
        cout << endl;
    }
    return 0;
}
```

思考:能不能在交换途中求出逆序对个数?

练习:

1. U248368 输入一个数n 和 n个整数,$a_1, a_2 \ldots, a_n$，将这些整数按照从小到大排序 n <= 5e6,a_i <= 1e7

2. 51nod3236

桶排序

桶排序（英文：Bucket sort）是排序算法的一种，适用于待排序数据值域较大但分布比较均匀的情况，同样需要内置排序。
算法步骤:设置一个定量的数组当作空桶；遍历序列，并将元素一个个放到对应的桶中；对每个不是空的桶进行排序；从不是空的桶里把元素再放回原来的序列中。
稳定性:若内置函数是稳定的就是稳定的，一般使用插入排序
空间复杂度: $O(n)$
时间复杂度: $O(n+n^2/k+k)$

```
// C++ Version
const int N = 100010;

int n, w, a[N];
vector<int> bucket[N];

void insertion_sort(vector<int>& A) {
  for (int i = 1; i < A.size(); ++i) {
    int key = A[i];
    int j = i - 1;
    while (j >= 0 && A[j] > key) {
      A[j + 1] = A[j];
      --j;
    }
    A[j + 1] = key;
  }
}

void bucket_sort() {
  int bucket_size = w / n + 1;
  for (int i = 0; i < n; ++i) {
    bucket[i].clear();
  }
  for (int i = 1; i <= n; ++i) {
    bucket[a[i] / bucket_size].push_back(a[i]);
  }
  int p = 0;
  for (int i = 0; i < n; ++i) {
    insertion_sort(bucket[i]);
    for (int j = 0; j < bucket[i].size(); ++j) {
      a[++p] = bucket[i][j];
    }
  }
}
```

思考:能不能在交换途中求出逆序对个数?

希尔排序

希尔排序（英语：Shell sort），也称为缩小增量排序法，是 插入排序 的一种改进版本
由于插入排序的复杂度取决于序列的有序程度，所以希尔排序是先使得它相对有序，再进行最后的排序
算法步骤:排序对不相邻的记录进行比较和移动：

1. 将待排序序列分为若干子序列（每个子序列的元素在原始数组中间距相同）；
2. 对这些子序列进行插入排序；
3. 减小每个子序列中元素之间的间距，重复上述过程直至间距减少为 1。

稳定性:不稳定
算法复杂度: o(n$^{1.3)\sim o(n}$2)

```
template <typename T>
void shell_sort(T array[], int length) {
  int h = 1;
  while (h < length / 3) {
    h = 3 * h + 1;
  }
  while (h >= 1) {
    for (int i = h; i < length; i++) {
      for (int j = i; j >= h && array[j] < array[j - h]; j -= h) {
        std::swap(array[j], array[j - h]);
      }
    }
    h = h / 3;
  }
}
```

思考:能不能在交换途中求出逆序对个数?

排序方法	时间复杂度（平均）	时间复杂度（最坏）	时间复杂度（最好）	空间复杂度	稳定性
插入排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
希尔排序	$O(n^{1.3})$	$O(n^2)$	$O(n)$	$O(1)$	不稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
堆排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(1)$	不稳定
冒泡排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
快速排序	$O(nlog_2n)$	$O(n^2)$	$O(nlog_2n)$	$O(nlog_2n)$	不稳定
归并排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(n)$	稳定
计数排序	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(n+k)$	稳定
桶排序	$O(n+k)$	$O(n^2)$	$O(n)$	$O(n+k)$	稳定
基数排序	$O(n*k)$	$O(n*k)$	$O(n*k)$	$O(n+k)$	稳定

进阶数据结构

堆

堆是一棵**完全二叉树**，树中每个结点的值都不小于（或不大于）其左右孩子结点的值。

其中，如果父亲结点的值大于或等于孩子结点的值，那么称这样的堆为大顶堆，这时每个结点的值都是以它为根结点的子树的最大值;如果父亲结点的值小于或等于孩子结点的值，那么称这样的堆为小顶堆，这时每个结点的值都是以它为根结点的子树的最小值。

以下都以小根堆为例子，给出一个序列 1， 5，7，3，2，4。小根堆可能就长这样。

对于完全二叉树，我们是知道是可以用数组存储的，每个节点的编号从上到下从左到右依次增大，这样一个编号为x节点的左儿子编号是$x*2$，右节点的编号是$x*2+1$

Down操作

堆里面最重要的两个操作一个是Down操作。Down操作可以在$O(logn)$的复杂度调整堆的结构。
比如我现在堆的样子长下面这样子

它目前不满足我们小根堆的一个性质，所以我们要把 5 这个数字往底下放
那么 5 这个数字首先要和它的两个儿子 2，3做比较，如果我们把 5 和 3交换的话，如图

我们发现还是不满足堆的性质，所以我们一定要选择当前节点与最小的儿子节点进行交换。

最后再拿 5 和 4进行交换即完成了我们的交换操作。
我们的Down操作也就是不停的拿父亲节点和两个子节点进行比较交换用的！

Up操作

堆里最后一个重要操作也就是Up操作，大家通过名字就可以类比出来，Down是往下调整，Up就是往上调整。
假设现在的图长这样

它并不满足小根堆的性质，我们需要把最下边的 1 往上调整。
1 只需要和 3做比较，因为 3 已经比 5 小了，所以如果 1 比 3 小的话，一定比 5 小。
交换后长这样

然后 1 再和 2做比较完成交换。

利用两大操作完成其他操作

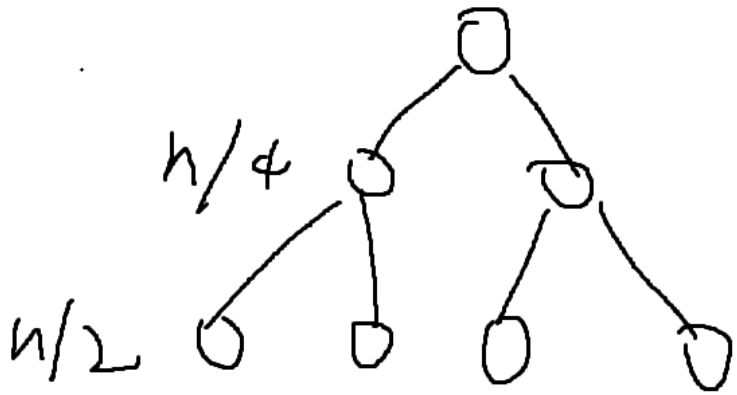
1. 修改堆里面其中一个数字，我们只需要利用Up和Down就可以完成调整
2. 往堆里面增加一个数字，首先我们要满足他是一个完全二叉树，刚好我们是使用数组来存储的，所以我们直接往数组最后一个位置进行添加，添加完后我们再使用Up操作来调整堆的结构即可。
3. 删除堆里的元素。首先我们要满足他是一个完全二叉树，刚好我们是使用数组来存储的，倘若我们删除的元素在数组最后一个位置就好了，所以我们就交换最后一个元素和我们要删除的那个元素，然后交换后进行Up和Down操作就可以调整堆的结构
4. 求当前堆里面的最小值，我们直接拿到数组第一个位置即可

构建堆

其实最简单构建堆的方法就是，不断地插入，插入后进行Up操作，但是这样的时间复杂度是$O(nlogn)$。
这里要讲一个$O(n)$的做法。

```
//n代表堆的大小
//heap[] 代表堆里面的元素
for(int i = n / 2;i >= 1; i--) down(i);
```

为什么这样子写是$O(n)$的呢?
假如这个堆是一个满二叉树，他的节点数是n，那么它最后一层的节点数大约是$n/2$倒数第二层的节点数大约是 $n/4$。



我们从倒数第二层开始一直到第一层往下Down，最后能完成树的结构调整。
对于倒数第二层来说往下Down一个点最多交换一次，那么所有点最多交换 $n/4$次，对于倒数第三层来说往下一个点Down最多交换两次，那么所有点最多交换 $2*n/8$次，以此类推最终交换的次数为
$n/4+2*n/8+3*n/16+\ldots+x*n/2^{x+1}$
设 $S_n=n/4+2*n/8+3*n/16+\ldots+x*n/2^{x+1}$

那么$2S_n = n/2 + 2*n/4 + 3*n/8 + .. + x*n/2^x$
$2S_n - S_n = n/2 + n/4 + n/8 + ... + n/2^x - x*n/2^{x+1}$约等于 n
所以最终这样建堆的复杂度就是$O(n)$

并查集

概论

定义：
并查集是一种树型的数据结构，用于处理一些不相交集合的合并及查询问题（即所谓的并、查）。比如说，我们可以用并查集来判断一个森林中有几棵树、某个节点是否属于某棵树等。

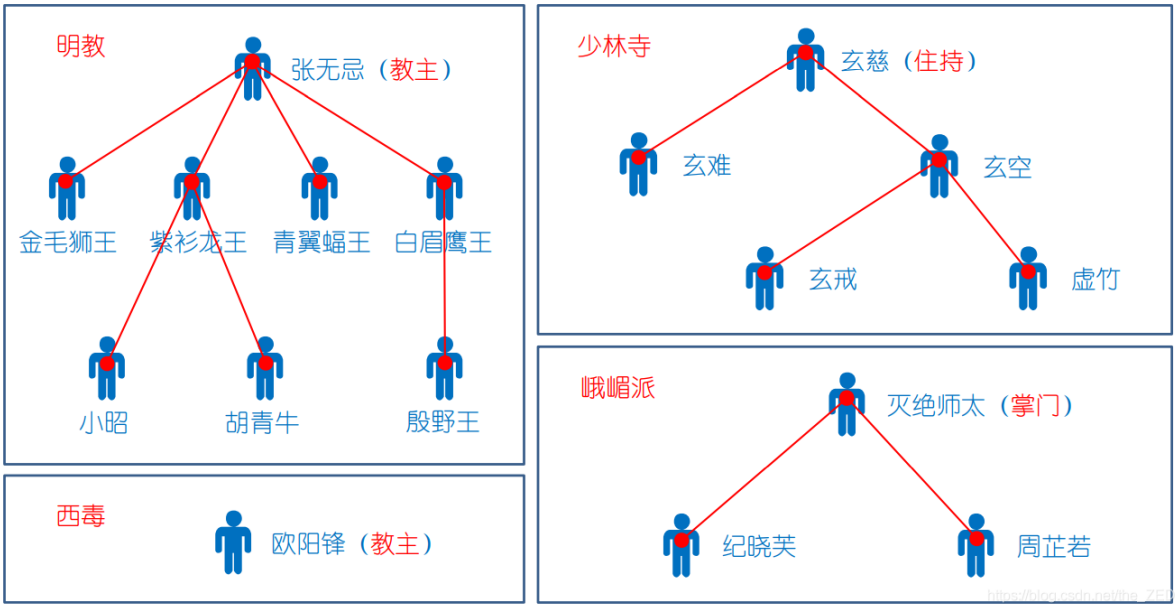
主要构成：
并查集主要由一个整型数组pre[ ]和两个函数find( )、join( )构成。
数组 pre[ ] 记录了每个点的前驱节点是谁，函数 find(x) 用于查找指定节点 x 属于哪个集合，函数 join(x,y) 用于合并两个节点 x 和 y 。

作用：
并查集的主要作用是求连通分支数（如果一个图中所有点都存在可达关系（直接或间接相连），则此图的连通分支数为1；如果此图有两大子图各自全部可达，则此图的连通分支数为2……）

并查集的现实意义

故事引入：
话说在江湖中散落着各式各样的大侠，他们怀揣着各自的理想和信仰在江湖中奔波。或是追求武林至尊，或是远离红尘，或是居庙堂之高，或是处江湖之远。尽管大多数人都安分地在做自己，但总有些人会因为彼此的信仰不同而聚众斗殴。因此，江湖上常年乱作一团，纷纷扰扰。
这样长期的混战，难免会打错人，说不定一刀就把拥有和自己相同信仰的队友给杀了。这该如何是好呢？于是，那些有着相同信仰的人们便聚在一起，进而形成了各种各样的门派，比如我们所熟知的“华山派”、“峨嵋派”、“,崆峒派”、“少林寺”、“明教”……这样一来，那些有着相同信仰的人们便聚在一起成为了朋友。以后再遇到要打架的事时，就不会打错人了。

但是新的问题又来了，原本互不相识的两个人如何辨别是否共属同一门派呢?
这好办！我们可以先在门派中选举一个“大哥”作为话事人（也就是掌门人，或称教主等）。这样一来，每当要打架的时候，决斗双方先自报家门，说出自己所在门派的教主名称，如果名称相同，就说明是自己人，就不必自相残杀了，否则才能进行决斗。于是，教主下令将整个门派划分为三六九等，使得整个门派内部形成一个严格的等级制度（即树形结构）。教主就是根节点，下面分别是二级、三级、……、N级队员。每个人只需要记住自己的上级名称，以后遇到需要辨别敌友的情况时，只需要一层层往上询问（网上询问）就能知道是否是同道中人了。



数据结构的角度来看：
由于我们的重点是在关注两个人是否连通，因此他们具体是如何连通的，内部结构是怎样的，甚至根节点是哪个（即教主是谁），都不重要。所以并查集在初始化时，教主可以随意选择（就不必再搞什么武林大会了），只要能分清敌友关系就行。

备注：上面所说的“教主”在教材中被称为“代表元”。
即：用集合中的某个元素来代表这个集合，则该元素称为此集合的代表元。

find( )函数的定义与实现

故事引入：

子夜，小昭于骊山下快马送信，发现一头戴竹笠之人立于前方，其形似黑蝠，倒挂于树前，甚惧，正系拔剑之时，只听四周悠悠传来：“如此夜深，姑凉竟敢擅闯明教，何不下坐陪我喝上一盅？”。小昭听闻，后觉此人乃明教四大护法之一的青翼蝠王韦一笑，答道：“在下小昭，乃紫衫龙王之女”。蝠王轻惕，急问道：“尔等既为龙王之女，故同为明教中人。敢问阁下教主大名，若非本教中人，于明教之地肆意走动那可是死罪！”。小昭吓得赶紧打了个电话问龙王：“龙王啊，咱教主叫啥名字来着？”，龙王答道：“吾教主乃张无忌也！”，小昭遂答蝠王：“张无忌！”。蝠王听后，抱拳请礼以让之。
在上面的情境中，小昭向他的上级（紫衫龙王）请示教主名称，龙王在得到申请后也向他的上级（张无忌）请示教主名称，此时由于张无忌就是教主，因此他直接反馈给龙王教主名称是“张无忌”。同理，青翼蝠王也经历了这一请示过程。
在并查集中，用于查询各自的教主名字的函数就是我们的find()函数。find(x)的作用是用于查找某个人所在门派的教主，换言之就是用于对某个给定的点x，返回其所属集合的代表。

实现：

首先我们需要定义一个数组：int pre[1000]；（数组长度依题意而定）。这个数组记录了每个人的上级是谁。这些人从0或1开始编号（依题意而定）。比如说pre[16]=6就表示16号的上级是6号。如果一个人的上级就是他自己，那说明他就是教主了，查找到此结束。也有孤家寡人自成一派的，比如欧阳锋，那么他的上级就是他自己。
每个人都只认自己的上级。比如小昭只知道自己的上级是紫衫龙王。教主是谁？不认识！要想知道自己教主的名称，只能一级级查上去。因此你可以视find(x)这个函数就是找教主用的。
下面给出这个函数的具体实现：

```
int find(int x)                 //查找x的教主{
    while(pre[x] != x)          //如果x的上级不是自己（则说明找到的人不是教主）
        x = pre[x];             //x继续找他的上级，直到找到教主为止
    return x;                   //教主驾到~~~
}
```

join( )函数的定义与实现

故事引入：
虚竹和周芷若是我个人非常喜欢的两个人物，他们的教主分别是玄慈方丈和灭绝师太，但是显然这两个人属于不同门派，但是我又不想看到他们打架。于是，我就去问了下玄慈和灭绝：“你看你们俩都没头发，要不就做朋友吧”。他们俩看在我的面子上同意了，这一同意非同小可，它直接换来了少林和峨眉的永世和平。

实现：
在上面的情景中，两个已存的不同门派就这样完成了合并。这么重大的变化，要如何实现？要改动多少地方？其实很简单，我对玄慈方丈说：“大师，麻烦你把你的上级改为灭绝师太吧。这样一来，两派原先所有人员的教主就都变成了师太，于是下面的人们也就不会打起来了！反正我们关心的只是连通性，门派内部的结构不要紧的”。玄慈听后立刻就不乐意了：“我靠，凭什么是我变成她手下呀，怎么不反过来？我抗议！”。抗议无效，我安排的，最大。反正谁加入谁效果是一样的，我就随手指定了一个，join()函数的作用就是用来实现这个的。

join(x,y)的执行逻辑如下：
1、寻找 x 的代表元（即教主）；
2、寻找 y 的代表元（即教主）；
3、如果 x 和 y 不相等，则随便选一个人作为另一个人的上级，如此一来就完成了 x 和 y 的合并。
下面给出这个函数的具体实现：

```
void join(int x,int y)                      //我想让虚竹和周芷若做朋友
{
    int fx=find(x), fy=find(y);             //虚竹的老大是玄慈，芷若MM的老大是灭绝
    if(fx != fy)                            //玄慈和灭绝显然不是同一个人
        pre[fx]=fy;                         //方丈只好委委屈屈地当了师太的手下啦
}
```

路径压缩算法之一（优化find( )函数）

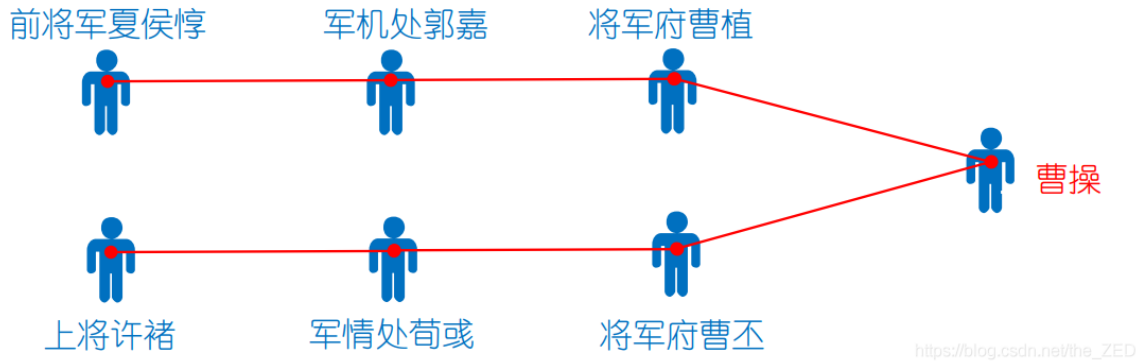
问题引入：

前面介绍的 join(x,y) 实际上为我们提供了一个将不同节点进行合并的方法。通常情况下，我们可以结合着循环来将给定的大量数据合并成为若干个更大的集合（即并查集）。但是问题也随之产生，我们来看这段代码：

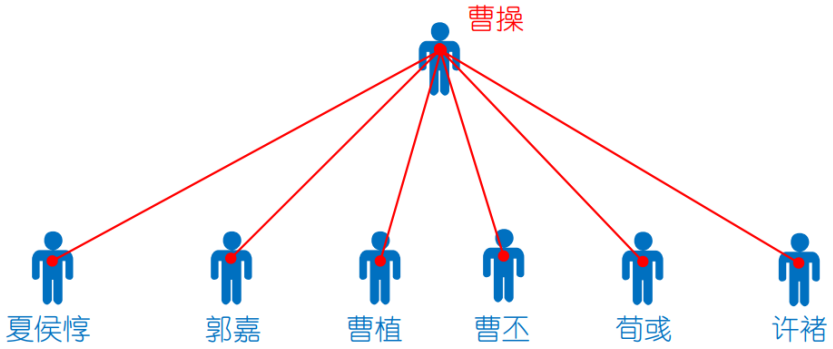
```
if(fx != fy)
    pre[fx]=fy;
```

这里并没有明确谁是谁的前驱（上级）的规则，而是我直接指定后面的数据作为前面数据的前驱（上级）。那么这样就导致了最终的树状结构无法预计，即有可能是良好的 n 叉树，也有可能是单支树结构（一字长蛇形）。试想，如果最后真的形成单支树结构，那么它的效率就会及其低下（树的深度过深，那么查询过程就必然耗时）。
而我们最理想的情况就是所有人的直接上级都是教主，这样一来整个树的结构就只有两级，此时查询教主只需要一次。因此，这就产生了路径压缩算法。
设想这样一个场景：两个互不相识的大将夏侯惇和许褚碰面了，他们都互相看不惯，想揍对方。于是按

照江湖规矩，先打电话问自己的上级：“你是不是教主？”上级说：“我不是呀，我的上级是***，我帮你问一下。”就这样两个人一路问下去，直到最终发现他们的教主都是曹操。具体结构如下：



“失礼失礼，原来是自己人，在下军机处前将军夏侯惇!”
“幸会幸会，不打不相识嘛，在下军情处上将许褚！”
紧接着，两人便高高兴兴地手拉手喝酒去了。
“等等等等，两位同学请留步，还有事情没完成呢！”我叫住他俩：“还要做路径压缩！”
两人醒悟，夏侯惇赶紧打电话给他的上级郭嘉：“军师啊，我查过了，我们的教主都是曹丞相。不如我们直接拜在丞相手下吧，省得级别太低，以后找起来太麻烦。”郭嘉答道：“所言甚是”。
许褚接着也打电话给刚才拜访过的荀彧，做了同样的事。于是此时，整个曹操阵营的结构如下所示：



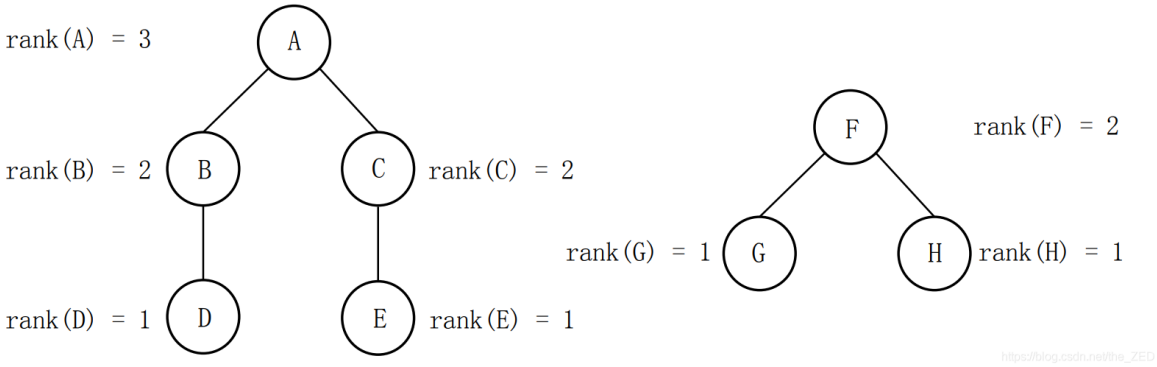
实现：
从上面的查询过程中不难看出，当从某个节点出发去寻找它的根节点时，我们会途径一系列的节点（比如上面的例子中，从夏侯惇→郭嘉→曹植→曹操），在这些节点中，除了根节点外（即曹操），其余所有节点（即夏侯惇、郭嘉、曹植）都需要更改直接前驱为曹操。
因此，基于这样的思路，我们可以通过递归的方法来逐层修改返回时的某个节点的直接前驱（即pre[x]的值）。简单说来就是将x到根节点路径上的所有点的pre（上级）都设为根节点。下面给出具体的实现代码：

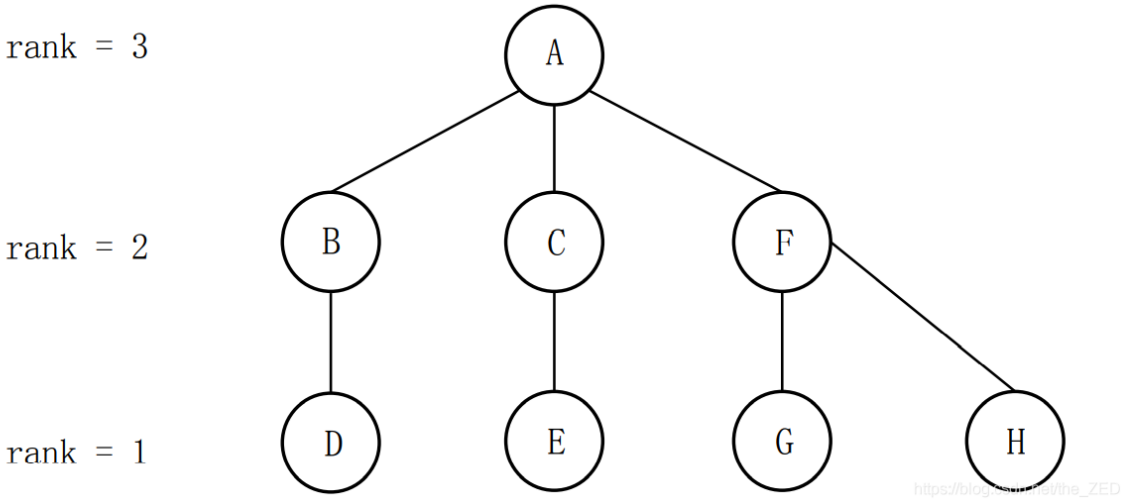
```
int find(int x)                     //查找结点 x的根结点 {
    if(pre[x] == x) return x;       //递归出口：x的上级为 x本身，即 x为根结点
    return pre[x] = find(pre[x]);   //此代码相当于先找到根结点 rootx，然后pre[x]=rootx
}
```

路径压缩算法之二（加权合并）

备注：
不要一看到“加权标记”这种比较高大上的名词就害怕，其实他也是属于路径压缩算法，不同的是其思想更加高级（更不容易想到）罢了。不过我还是说一下，掌握了前面几点内容就能应对绝大多数ACM问题，下面的“加权标记法”供有兴趣的同学学习交流。

主要思路：
加权标记法需要将树中所有节点都增设一个权值，用以表示该节点所在树中的高度（比如用rank[x]=3表示 x 节点所在树的高度为3）。这样一来，在合并操作的时候就能通过这个权值的大小来决定谁当谁的上级（玄慈哭了："正义终会来到，但永不会缺席"）。
在合并操作的时候，假设需要合并的两个集合的代表元（教主）分别为 x 和 y，则只需要令pre[x] = y 或者pre[y] = x 即可。但我们为了使合并后的树不产生退化（即：使树中左右子树的深度差尽可能小），那么对于每一个元素 x ，增设一个rank[x]数组，用以表达子树 x 的高度。在合并时，如果rank[x] < rank[y]，则令pre[x] = y；否则令pre[y] = x。
举个例子，我们对以A，F为代表元的集合进行合并操作（如下图所示）：





可以看出，合并前两个树的最大高度为3，合并后依然是3，这也就达到了我们的目的。但如果令pre[A]=F，那么就会使得合并后的树的总高度增加，这里我就不上图了，同学们不信可以自己画出来看看。
我们常说，鱼和熊掌不可兼得，同理，时间复杂度和空间复杂度也很难兼得。由于给每个节点都增加了一个权值来标记其在树中的高度，这也就意味着需要额外的数据结构来存放权重信息，所以这将导致额外的空间开销。

实现：
加权标记法的核心在于对rank数组的逻辑控制，其主要的情况有：
1、如果rank[x] < rank[y]，则令pre[x] = y；
2、如果rank[x] == rank[y]，则可任意指定上级；
3、如果rank[x] > rank[y]，则令pre[y] = x；
在实际写代码时，为了使代码尽可能简洁，我们可以将第1点单独作为一个逻辑选择，然后将2、3点作为另一个选择（反正第2点任意指定上级嘛），所以具体的代码如下：

```
void union(int x,int y){
    x=find(x);                              //寻找 x的代表元
    y=find(y);                              //寻找 y的代表元
    if(x==y) return ;                       //如果 x和 y的代表元一致，说明他们共属同一集合，
则不需要合并，直接返回；否则，执行下面的逻辑
    if(rank[x]>rank[y]) pre[y]=x;           //如果 x的高度大于 y，则令 y的上级为 x
    else                                    //否则{
        if(rank[x]==rank[y]) rank[y]++; //如果 x的高度和 y的高度相同，则令 y的高度加1
        pre[x]=y;                           //让 x的上级为 y
    }
}
```

单调队列

引入

题目描述

复制Markdown 展开

有一个长为 n 的序列 a，以及一个大小为 k 的窗口。现在这个从左边开始向右滑动，每次滑动一个单位，求出每次滑动后窗口中的最大值和最小值。

例如：

The array is $[1,3,-1,-3,5,3,6,7]$, and $k=3$。

Window position	Minimum value	Maximum value
[1 3 -1] -3 5 3 6 7	-1	3
1 [3 -1 -3] 5 3 6 7	-3	3
1 3 [-1 -3 5] 3 6 7	-3	5
1 3 -1 [-3 5 3] 6 7	-3	5
1 3 -1 -3 [5 3 6] 7	3	6
1 3 -1 -3 5 [3 6 7]	3	7

输入格式

输入一共有两行，第一行有两个正整数 n,k。 第二行 n 个整数，表示序列 a

输出格式

输出共两行，第一行为每次窗口滑动的最小值
第二行为每次窗口滑动的最大值

输入输出样例

【数据范围】
对于 50%的数据，$1 \le n \le 10^5$；
对于 100% 的数据，$1 \le k \le n \le 10^6$，$a_i \in [-2^{31}, 2^{31})$
思路：最暴力的想法很简单，对于每一段 $i \sim i+k-1$ 的序列，逐个比较来找出最大值（和最小值），时间复杂度约为 $O(n \times k)$。很显然，这其中进行了大量重复工作，除了开头 $k-1$ 个和结尾 $k-1$ 个数之外，每个数都进行了 k 次比较，而题中 100% 的数据为 $n \le 1000000$，当 k 稍大的情况下，显然会 TLE。

这时所用到的就是单调队列了。

定义

顾名思义，单调队列的重点分为「单调」和「队列」。

「单调」指的是元素的「规律」——递增（或递减）。

「队列」指的是元素只能从队头和队尾进行操作。

Ps. 单调队列中的 "队列" 与正常的队列有一定的区别，稍后会提到

例题分析

解释

有了上面「单调队列」的概念，很容易想到用单调队列进行优化。

要求的是每连续的 k 个数中的最大（最小）值，很明显，当一个数进入所要 "寻找" 最大值的范围中时，若这个数比其前面（先进队）的数要大，显然，前面的数会比这个数先出队且不再可能是最大值。

也就是说——当满足以上条件时，可将前面的数 "弹出"，再将该数真正 push 进队尾。

这就相当于维护了一个递减的队列，符合单调队列的定义，减少了重复的比较次数，不仅如此，由于维护出的队伍是查询范围内的且是递减的，队头必定是该查询区域内的最大值，因此输出时只需输出队头即可。

显而易见的是，在这样的算法中，每个数只要进队与出队各一次，因此时间复杂度被降到了 $O(N)$。

而由于查询区间长度是固定的，超出查询空间的值再大也不能输出，因此还需要 site 数组记录第 i 个队中的数在原数组中的位置，以弹出越界的队头。

过程

例如我们构造一个单调递增的队列会如下：

原序列为：

```
1 3 -1 -3 5 3 6 7
```

因为我们始终要维护队列保证其 **递增** 的特点，所以会有如下的事情发生：

操作	队列状态
1 入队	`{1}`
3 比 1 大，3 入队	`{1 3}`
-1 比队列中所有元素小，所以清空队列 -1 入队	`{-1}`
-3 比队列中所有元素小，所以清空队列 -3 入队	`{-3}`
5 比 -3 大，直接入队	`{-3 5}`
3 比 5 小，5 出队，3 入队	`{-3 3}`
-3 已经在窗体外，所以 -3 出队；6 比 3 大，6 入队	`{3 6}`
7 比 6 大，7 入队	`{3 6 7}`

```
#include <cstdio>
```

```
#include <cstdlib>
#include <cstring>
#include <iostream>
#define maxn 1000100
using namespace std;
int q[maxn], a[maxn];
int n, k;

void getmin() {  // 得到这个队列里的最小值，直接找到最后的就行了
  int head = 1, tail = 0;
  for (int i = 1; i < k; i++) {
    while (head <= tail && a[q[tail]] >= a[i]) tail--;
    q[++tail] = i;
  }
  for (int i = k; i <= n; i++) {
    while (head <= tail && a[q[tail]] >= a[i]) tail--; //如果新元素比之前的还要小，那么就把旧元素踢掉
    q[++tail] = i;
    while (q[head] <= i - k) head++; //判断是否超出K了
    printf("%d ", a[q[head]]);
  }
}

void getmax() {  // 和上面同理
  int head = 1, tail = 0;
  for (int i = 1; i < k; i++) {
    while (head <= tail && a[q[tail]] <= a[i]) tail--;
    q[++tail] = i;
  }
  for (int i = k; i <= n; i++) {
    while (head <= tail && a[q[tail]] <= a[i]) tail--;
    q[++tail] = i;
    while (q[head] <= i - k) head++;
    printf("%d ", a[q[head]]);
  }
}

int main() {
  scanf("%d%d", &n, &k);
  for (int i = 1; i <= n; i++) scanf("%d", &a[i]);
  getmin();
  printf("\n");
  getmax();
  printf("\n");
  return 0;
}
```

老板需要你帮忙浇花。给出N滴水的坐标，y表示水滴的高度，x表示它下落到x轴的位置。

每滴水以每秒1个单位长度的速度下落。你需要把花盆放在x轴上的某个位置，使得从被花盆接着的第1滴水开始，到被花盆接着的最后1滴水结束，之间的时间差至少为D。

我们认为，只要水滴落到x轴上，与花盆的边沿对齐，就认为被接住。给出N滴水的坐标和D的大小，请算出最小的花盆的宽度W。

输入格式

第一行2个整数 N 和 D。

第2.. N+1行每行2个整数，表示水滴的坐标(x,y)。

输出格式

仅一行1个整数，表示最小的花盆的宽度。如果无法构造出足够宽的花盆，使得在D单位的时间接住满足要求的水滴，则输出-1。

输入输出样例

$1 \le N \le 100000, 1 \le D \le 1000000, 0 \le x, y \le 10^6$

思路1：将所有水滴按照 x 坐标排序之后，题意可以转化为求一个 x 坐标差最小的区间使得这个区间内 y 坐标的最大值和最小值之差至少为 D。我们发现这道题和上一道例题有相似之处，就是都与一个区间内的最大值最小值有关，但是这道题区间的大小不确定，而且区间大小本身还是我们要求的答案。对于区间来说，区间越大区间的最大值一定越大，区间的最小值一定越小。我们要求最大值 - 最小值 >= D，倘若花盆的长度小的不满足答案，我们肯定扩大。当满足条件后，我们肯定缩小去尝试更小的，很容易想到这就是一个二分答案。对于二分答案，我们如何去查看那么多区间是否有一个区间满足最大值 - 最小值 >= D，那就是使用单调队列即可。

思路2：我们依然可以使用一个递增，一个递减两个单调队列在 R 不断后移时维护 $[L, R]$ 内的最大值和最小值，不过此时我们发现，如果 L 固定，那么 $[L, R]$ 内的最大值只会越来越大，最小值只会越来越小，所以设 $f(R) = \max[L, R] - \min[L, R]$，则 $f(R)$ 是个关于 R 的递增函数，故 $f(R) \ge D \Rightarrow f(r) \ge D, R < r \le N$。这说明对于每个固定的 L，向右第一个满足条件的 R 就是最优答案。
所以我们整体求解的过程就是，先固定 L，从前往后移动 R，使用两个单调队列维护 $[L, R]$ 的最值。当找到了第一个满足条件的 R，就更新答案并将 L 也向后移动。随着 L 向后移动，两个单调队列都需及时弹出队头。这样，直到 R 移到最后，每个元素依然是各进出队列一次，保证了 $O(n)$ 的时间复杂度。

```
#include <bits/stdc++.h>
using namespace std;
const int N = 100005;
typedef long long ll;
int mxq[N], mnq[N];
int D, ans, n, hx, rx, hn, rn;

struct la {
  int x, y;

  bool operator<(const la &y) const { return x < y.x; }
} a[N];

int main() {
  scanf("%d%d", &n, &D);
  for (int i = 1; i <= n; ++i) {
    scanf("%d%d", &a[i].x, &a[i].y);
```

```
  }
  sort(a + 1, a + n + 1);
  hx = hn = 1;
  ans = 2e9;
  int L = 1;
  for (int i = 1; i <= n; ++i) {
    while (hx <= rx && a[mxq[rx]].y < a[i].y) rx--;
    mxq[++rx] = i;
    while (hn <= rn && a[mnq[rn]].y > a[i].y) rn--;
    mnq[++rn] = i;
    while (L <= i && a[mxq[hx]].y - a[mnq[hn]].y >= D) {
      ans = min(ans, a[i].x - a[L].x);
      L++;
      while (hx <= rx && mxq[hx] < L) hx++;
      while (hn <= rn && mnq[hn] < L) hn++;
    }
  }
  if (ans < 2e9)
    printf("%d\n", ans);
  else
    puts("-1");
  return 0;
}
```

单调栈

Trie树

哈希表

KMP

STL模板

RMQ问题

进阶前缀和和差分练习

进阶递推与递归练习

进阶二分练习

进阶图论

图论知识

二分图

欧拉图

有向无环图

连通图

强连通图

重连通图

拓扑排序

单源最短路之bellman - ford

单源最短路之spfa

单源最短路之Dijkstra

多源最短路的建边方式

多源最短路之Floyd

最小生成树Prim

最小生成树Kruskal

二分图

二分图匹配

搜索

多源BFS

DFS之连通性

DFS之搜索顺序

DFS之剪枝与优化

迭代加深

双向DFS

进阶数论

欧拉定理和欧拉函数

费马小定理

扩展欧几里得算法

威尔逊定理

裴蜀定理

中国剩余定理

容斥定理

线性代数

进阶动态规划

数字三角模型

最长上升子序列模型

记忆化搜索

背包模型

状态压缩dp

树形dp

组合数学

更进阶数据结构

进阶并查集

权值并查集

顾名思义，通过权值来设置特殊性质的并查集。

洛谷 / 题目列表 / 题目详情

P2024 [NOI2001] 食物链

提交答案 加入题单 复制题目

题目描述

复制Markdown 展开

动物王国中有三类动物 A,B,C，这三类动物的食物链构成了有趣的环形。A 吃 B，B 吃 C，C 吃 A。

现有 N 个动物，以 $1\sim N$ 编号。每个动物都是 A,B,C 中的一种，但是我们并不知道它到底是哪一种。

有人用两种说法对这 N 个动物所构成的食物链关系进行描述：

- 第一种说法是 `1 X Y`，表示 X 和 Y 是同类。
- 第二种说法是 `2 X Y`，表示 X 吃 Y。

此人对 N 个动物，用上述两种说法，一句接一句地说出 K 句话，这 K 句话有的是真的，有的是假的。当一句话满足下列三条之一时，这句话就是假话，否则就是真话。

- 当前的话与前面的某些真的话冲突，就是假话；
- 当前的话中 X 或 Y 比 N 大，就是假话；
- 当前的话表示 X 吃 X，就是假话。

你的任务是根据给定的 N 和 K 句话，输出假话的总数。

输入格式

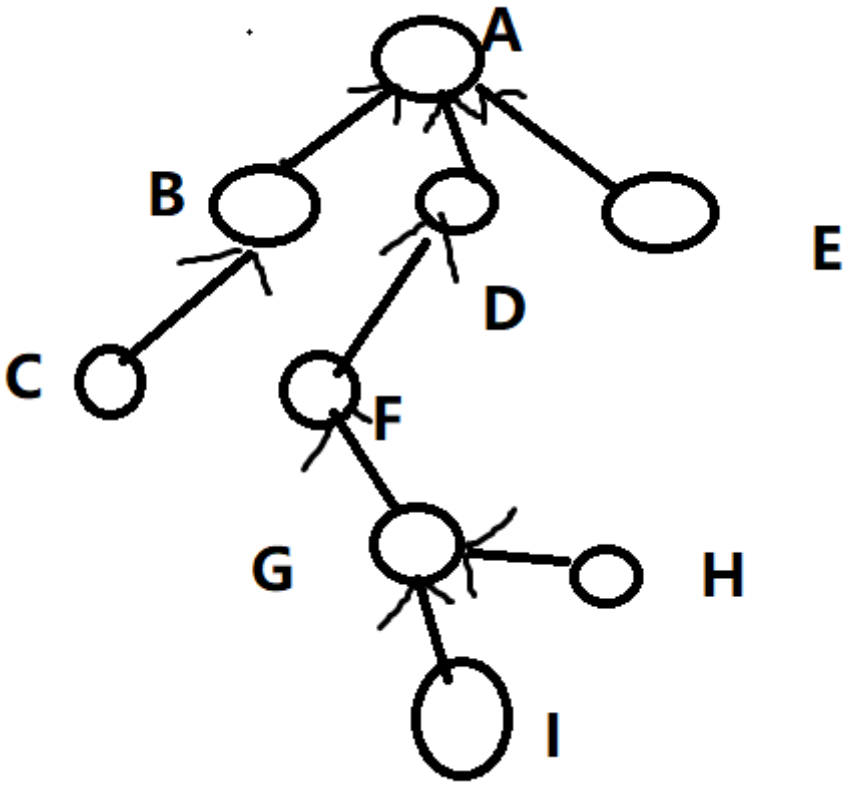
第一行两个整数，N,K，表示有 N 个动物，K 句话。

第二行开始每行一句话（按照题目要求，见样例）

输出格式

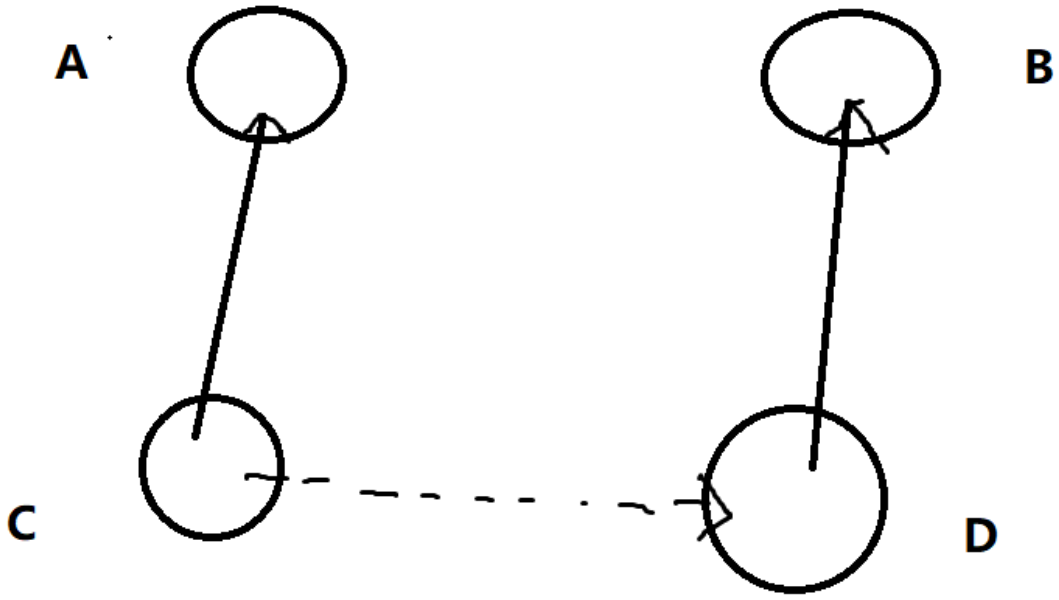
一行，一个整数，表示假话的总数。

咋们通过以上题目来进行讲解。
我们思考A -> B B-> C 的话 一定是能推出 C->A，所以ABC在一个集合里面，我们只需要知道其中两个的关系，就能推导第三个。



我们假设这是一些动物的关系，我们可以通过图片看出来，A被B,D,E吃，C,F被A吃，G是A的同类。

那么我们发现，若我们当前动物层数是X，那么与我们为同类的即为，X + 3 * n层，吃我们的是 X + 1 + 3 * n层，被我们吃的是 X + 2 + 3 * n层
对于3取模的话就只剩3种情况，0，1，2。所以最后，我们只需要维护每个点到他祖宗的深度就可以了。
那么主要难点的话还是合并，我们该怎么合并呢?



假设C，D同类我们现在要合并他们的祖宗节点A，C
由于 AB + AC == BD + CD 所以　AC + AB ≡ BD (mod 3)
也就是说 (AC + AB - BD) mod 3 == 0
所以AB = BD - AC

```
//这是求深度的关键代码
int find(int x){
    if (p[x] != x){
        int t = find(p[x]);
        d[x] += d[p[x]];
        p[x] = t;
    }
    return p[x];
}
```

总体食物链代码

```
#include <iostream>

using namespace std;

const int N = 50010;

int n, m;
int p[N], d[N];

int find(int x){
    if (p[x] != x){
        int t = find(p[x]);
        d[x] += d[p[x]];
        p[x] = t;
    }
    return p[x];
}

int main(){
    scanf("%d%d", &n, &m);
    for (int i = 1; i <= n; i ++ ) p[i] = i;
    int res = 0;
    while (m -- ){
        int t, x, y;
        scanf("%d%d%d", &t, &x, &y);
        if (x > n || y > n) res ++ ;
        else{
            int px = find(x), py = find(y);
            if (t == 1){
                if (px == py && (d[x] - d[y]) % 3) res ++ ;
                else if (px != py){
                    p[px] = py;
                    d[px] = d[y] - d[x];
                }
            }else{
                if (px == py && (d[x] - d[y] - 1) % 3) res ++ ;
                else if (px != py){
                    p[px] = py;
                    d[px] = d[y] + 1 - d[x];
                }
            }
        }
    }
    printf("%d\n", res);
```

```
    return 0;
}
```

树状数组

线段树

更进阶图论

单源最短路的建边方式

1. **1对1**

题目描述

复制Markdown 展开

有一个 n 个点 m 条边的无向图，请求出从 s 到 t 的最短路长度。

输入格式

第一行四个正整数 n, m, s, t。 接下来 m 行，每行三个正整数 u, v, w，表示一条连接 u, v，长为 w 的边。

输出格式

输出一行一个整数，表示答案。

输入输出样例

输入 #1 复制

```
7 11 5 4
2 4 2
1 4 3
7 2 2
3 4 3
5 7 5
7 3 3
6 1 1
6 3 4
2 4 3
5 6 3
7 2 1
```

输出 #1 复制

```
7
```

说明/提示

【数据范围】
对于 100% 的数据，$1 \le n \le 2500$，$1 \le m \le 6200$，$1 \le w \le 1000$。

2. **1对多**

战争时期，前线有 n 个哨所，每个哨所可能会与其他若干个哨所之间有通信联系。

信使负责在哨所之间传递信息，当然，这是要花费一定时间的（以天为单位）。

指挥部设在第一个哨所。

当指挥部下达一个命令后，指挥部就派出若干个信使向与指挥部相连的哨所送信。

当一个哨所接到信后，这个哨所内的信使们也以同样的方式向其他哨所送信。**信在一个哨所内停留的时间可以忽略不计。**

直至所有 n 个哨所全部接到命令后，送信才算成功。

因为准备充足，每个哨所内都安排了足够的信使（如果一个哨所与其他 k 个哨所有通信联系的话，这个哨所内至少会配备 k 个信使）。

现在总指挥请你编一个程序，计算出完成整个送信过程最短需要多少时间。

输入格式

第 1 行有两个整数 n 和 m，中间用 1 个空格隔开，分别表示有 n 个哨所和 m 条通信线路。

第 2 至 $m+1$ 行：每行三个整数 i、j、k，中间用 1 个空格隔开，表示第 i 个和第 j 个哨所之间存在 **双向** 通信线路，且这条线路要花费 k 天。

输出格式

一个整数，表示完成整个送信过程的最短时间。

如果不是所有的哨所都能收到信，就输出-1。

数据范围

$1 \leq n \leq 100$,
$1 \leq m \leq 200$,
$1 \leq k \leq 1000$

3. 多对多

农夫John发现了做出全威斯康辛州最甜的黄油的方法：糖。

把糖放在一片牧场上，他知道 N 只奶牛会过来舔它，这样就能做出能卖好价钱的超甜黄油。

当然，他将付出额外的费用在奶牛上。

农夫John很狡猾，就像以前的巴甫洛夫，他知道他可以训练这些奶牛，让它们在听到铃声时去一个特定的牧场。

他打算将糖放在那里然后下午发出铃声，以至他可以在晚上挤奶。

农夫John知道每只奶牛都在各自喜欢的牧场（一个牧场不一定只有一头牛）。

给出各头牛在的牧场和牧场间的路线，找出使所有牛到达的路程和最短的牧场（他将把糖放在那）。

数据保证至少存在一个牧场和所有牛所在的牧场连通。

输入格式

第一行: 三个数：奶牛数 N，牧场数 P，牧场间道路数 C。

第二行到第 N+1 行: 1 到 N 头奶牛所在的牧场号。

第 N+2 行到第 N+C+1 行：每行有三个数：相连的牧场A、B，两牧场间距 D，当然，连接是双向的。

输出格式

共一行，输出奶牛必须行走的最小的距离和。

数据范围

$1 \leq N \leq 500$,
$2 \leq P \leq 800$,
$1 \leq C \leq 1450$,
$1 \leq D \leq 255$

4.反向建图

在 n 个人中，某些人的银行账号之间可以互相转账。

这些人之间转账的手续费各不相同。

给定这些人之间转账时需要从转账金额里扣除百分之几的手续费，请问 A 最少需要多少钱使得转账后 B 收到 100 元。

输入格式

第一行输入两个正整数 n, m，分别表示总人数和可以互相转账的人的对数。

以下 m 行每行输入三个正整数 x, y, z，表示标号为 x 的人和标号为 y 的人之间互相转账需要扣除 $z\%$ 的手续费 ($z < 100$)。

最后一行输入两个正整数 A, B。

数据保证 A 与 B 之间可以直接或间接地转账。

输出格式

输出 A 使得 B 到账 100 元最少需要的总费用。

精确到小数点后 8 位。

数据范围

$1 \leq n \leq 2000$,
$m \leq 10^5$

5.建立虚拟原点，减少边的数量

> H 城是一个旅游胜地，每年都有成千上万的人前来观光。
>
> 为方便游客，巴士公司在各个旅游景点及宾馆，饭店等地都设置了巴士站并开通了一些单程巴士线路。
>
> 每条单程巴士线路从某个巴士站出发，依次途经若干个巴士站，最终到达终点巴士站。
>
> 一名旅客最近到 H 城旅游，他很想去 S 公园游玩，但如果从他所在的饭店没有一路巴士可以直接到达 S 公园，则他可能要先乘某一路巴士坐几站，再下来换乘同一站台的另一路巴士，这样换乘几次后到达 S 公园。
>
> 现在用整数 $1,2,\ldots N$ 给 H 城的所有的巴士站编号，约定这名旅客所在饭店的巴士站编号为 1，S 公园巴士站的编号为 N。
>
> 写一个程序，帮助这名旅客寻找一个最优乘车方案，使他在从饭店乘车到 S 公园的过程中换乘的次数最少。
>
> **输入格式**
>
> 第一行有两个数字 M 和 N，表示开通了 M 条单程巴士线路，总共有 N 个车站。
>
> 从第二行到第 $M+1$ 行依次给出了第 1 条到第 M 条巴士线路的信息，其中第 $i+1$ 行给出的是第 i 条巴士线路的信息，从左至右按运行顺序依次给出了该线路上的所有站号，相邻两个站号之间用一个空格隔开。
>
> **输出格式**
>
> 共一行，如果无法乘巴士从饭店到达 S 公园，则输出 `NO`，否则输出最少换乘次数，换乘次数为 0 表示不需换车即可到达。
>
> **数据范围**
>
> $1 \le M \le 100$,
> $2 \le N \le 500$

6.适量的增加复杂度来换取正确答案（枚举）

> 在那里他和酋长的女儿相爱了，于是便向酋长去求亲。
>
> 酋长要他用 10000 个金币作为聘礼才答应把女儿嫁给他。
>
> 探险家拿不出这么多金币，便请求酋长降低要求。
>
> 酋长说："嗯，如果你能够替我弄到大祭司的皮袄，我可以只要 8000 金币。如果你能够弄来他的水晶球，那么只要 5000 金币就行了。"
>
> 探险家就跑到大祭司那里，向他要求皮袄或水晶球，大祭司要他用金币来换，或者替他弄来其他的东西，他可以降低价格。
>
> 探险家于是又跑到其他地方，其他人也提出了类似的要求，或者直接用金币换，或者找到其他东西就可以降低价格。
>
> 不过探险家没必要用多样东西去换一样东西，因为不会得到更低的价格。
>
> 探险家现在很需要你的帮忙，让他用最少的金币娶到自己的心上人。
>
> 另外他要告诉你的是，在这个部落里，等级观念十分森严。
>
> 地位差距超过一定限制的两个人之间不会进行任何形式的直接接触，包括交易。
>
> 他是一个外来人，所以可以不受这些限制。
>
> 但是如果他和某个地位较低的人进行了交易，地位较高的的人不会再和他交易，他们认为这样等于是间接接触，反过来也一样。
>
> 因此你需要在考虑所有的情况以后给他提供一个最好的方案。
>
> 为了方便起见，我们把所有的物品从 1 开始进行编号，酋长的允诺也看作一个物品，并且编号总是 1。
>
> 每个物品都有对应的价格 P，主人的地位等级 L，以及一系列的替代品 T_i 和该替代品所对应的"优惠" V_i。
>
> 如果两人地位等级差距超过了 M，就不能"间接交易"。
>
> 你必须根据这些数据来计算出探险家最少需要多少金币才能娶到酋长的女儿。

单源最短路的综合应用

单源最短路的扩展应用

负环问题

Floyd算法的应用

最小生成树进阶

最小生成树的扩展应用

差分约束算法

最近公共祖先算法

拓扑排序的进阶应用

欧拉回路和欧拉路径

更进阶数论

筛法的使用

快速幂

约数进阶

欧拉函数进阶

同余进阶

进阶线性代数

矩阵乘法

高斯消元

组合数学进阶

鸽巢原理

卡特兰数

容斥原理进阶

错排

更进阶动态规划

状态机模型

数位dp

单调队列优化dp

更更进阶图论

前置知识

求强连通分量算法

强连通分量的缩点算法

求割点割边

更更进阶数据结构

二叉平衡树之treap

二叉平衡树之splay

基环树